Computing Complex Events in an Event-driven and Logic-based Approach - (Demo)

Darko Anicic[1]
darko.anicic@fzi.de

Paul Fodor[2]
pfodor@cs.sunysb.edu

Roland Stühmer[1]
roland.stuehmer@fzi.de

Nenad Stojanovic[1]
nenad.stojanovic@fzi.de

[1]FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
[2]Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, U.S.A

ABSTRACT

In this paper we propose to demonstrate a logic-based and data-driven complex event processor. The event processor is called ETALIS (Event-driven Transaction Logic Inference System). ETALIS is based on decomposition of complex event patterns into patterns goals. Goals are asserted by declarative rules, which are executed in the backward chaining mode. A specific property of these rules is that they are event-driven, and allow reasoning over events relationships.

1. LOGICAL FRAMEWORK FOR EVENTS

We have developed a logic-based framework for Complex Event Processing called ETALIS (Event-driven Transaction Logic Inference System). ETALIS is an open source project[1]. Event patterns in the system are represented declaratively. The approach is established on decomposition of complex event patterns into *two-input intermediate events* (i.e., *goals*). The status of achieved goals at the current state shows the progress toward completeness of an event pattern. Goals are automatically asserted by rules as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or pattern.

We demonstrate the approach using an example for computing sequences of events. For example, the following rules (1)-(3) define event patterns based on sequences of events (e.g., $e1$ occurs when an event a is followed by an event b, followed by c). Figure 1 depicts a graph representation of events sequences for $e1$, $e2$ and $e3$ (a node with symbol $\otimes$ denotes a sequence). Additionally it shows that $e1$ is used to trigger both $action1$ and $action2$, while $e2$ and $e3$ trigger $action2$ and $action3$, respectively.

$$e1([T1,T6]) \leftarrow a([T1,T2]) \otimes b([T3,T4]) \otimes c([T5,T6]). \quad (1)$$
$$e2([T1,T6]) \leftarrow b([T1,T2]) \otimes c([T3,T4]) \otimes d([T5,T6]). \quad (2)$$
$$e3([T1,T4]) \leftarrow c([T1,T2]) \otimes d([T3,T4]). \quad (3)$$

[1]ETALIS project: http://code.google.com/p/etalis/

DEBS '09 Nashville, TN, USA

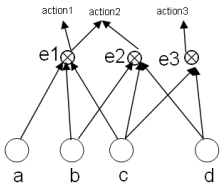

Figure 1: A sequences of events

Rules (1)-(3) specify complex event patterns $e1$, $e2$ and $e3$. In general, they are created by a user using *event operators* available from a particular event processing language. In our example, we are currently focusing only on a sequential composition. ETALIS further offers: conjunction of events ($\wedge$), concurrent operator ($|$), negation ($\neg$), disjunction ($\vee$), N-Times operator (*event^N*) etc. For the full list of implemented event operators, a reader is referred to [1]. Every event a, b is defined over a time interval $[T_1, T_2]$ with possible set of data terms $(X_1, X_2, ..., X_n)$. Event patterns contain data relevant for a reactive system[2]. The data of an event pattern is a set of terms which may be either a variable, a constant, or a function symbol.

In their present form, rules (1)-(3) are not convenient to be used for event-driven computation. These are rather a Prolog-style rules suitable for backward chaining evaluation. Such rules are understood as goals, which at certain time either can or cannot be proved by an inference engine. A provable goal means that a corresponding pattern has occurred, and vice versa. Note that such an evaluation is not appropriate for *event-driven* computation, as it does not detect events when they really occur. Rather it *proves* them at the time when corresponding goals (queries) are set. This is why this kind of event processing is called *query-driven*.

We have devised an algorithm that automatically transforms user defined event rules (e.g., rules (1)-(3)) into *event-driven backward chaining* rules [1]. The *event-driven backward chaining* rules are suitable for detection of events as soon as they occur. Further on, detection of events is focused on patterns of interest, rather than merely detecting irrelevant events. For further details related to the algorithm, a reader is referred to our technical report [1]. In the following we present first evaluation results from our implementation of that algorithm.

2. DEMONSTRATION

[2]We avoid representing data here due to space restrictions.

Complex events, detected by ETALIS, can be used for triggering a set of actions. As shown in Figure 1, $e1$ triggers $action1$ and $action2$, $e2$ and $e3$ activate $action2$ and $action3$, respectively. In our example, actions are simple (they only print its name with the time when an action has been triggered). In general case, our prototype supports more complex actions. For instance, an action can be composed of more concurrently running sub-actions which are synchronized by detected complex events and/or between themselves.

Figure 2 shows a listing after the execution of rules (1)-(3). We have assumed a situation where a sequence of atomic events of type a, b, c, d has occurred in time points $T_i = 1, 2, 3, 4$[3]. As a result, when c occurred (i.e., at $T_i = 3$), $action1$ and $action2$ have been triggered (i.e., with a detection interval $[1, 3]$ for $e1$). Similarly after occurrence of d, $action2$ and $action3$ have been activated as consequences of detection $e2$ and $e3$ (they only have different intervals as the time of detection for $e2$ and $e3$ is different too, Figure 2).

```
xsb -e "[load],init,ctr_comp('event_02')."
yes
| ?- execCTR(a([1,1])).

yes
| ?- execCTR(b([2,2])).

yes
| ?- execCTR(c([3,3])).
action1([1,3])
action2([1,3])

yes
| ?- execCTR(d([4,4])).
action2([2,4])
action3([3,4])

yes
| ?-
```

Figure 2: Detection of sequence of events and action triggering

Performance Results. We present first performance results of our prototype implementation. We have implemented algorithms that transform user defined event pattern rules into rules suitable for event-driven pattern detection, and execute events in XSB Prolog. All the test cases were run on a workstation with Pentium dual-core processor 2,4GHz CPU and 3GB memory running on Ubuntu Linux and XSB Prolog version 3.1. In our experiments we have used different operators to generate a few events in a block of complex patterns. The first experiment consists of six atomic events (i.e., e_1-e_6) and four complex events (i.e., $ce1_1$ - $ce1_4$)), see experiment rules (4) below. Further, we have been multiplying the number of event blocks to assess the throughput of input (atomic) events that can be handled (i.e., used for detection of complex patterns) by our prototype. The sequences are of depth between two and one million events.

$$\begin{array}{l} ce1_1 : -e_1 \otimes e_2. \\ ce1_2 : -e_2 \otimes e_3 \otimes e_4. \\ ce1_3 : -(e_3 \otimes e_5)|(e_4 \otimes e_6). \\ ce1_4 : -(e_2 \vee e_3) \wedge (ce1_2|\neg e_5). \end{array} \quad (4)$$

Figure 3 shows the experimental results obtained with the prototype. The test is repeated an increasing number of times. The maximum throughput that we got is in a range between 12600 and 13500 events/second. We tested for sizes of 10,000 blocks (4438ms), 50,000 (21360ms) and 500,000 (237950ms).

The memory consumption during the pattern matching process is very low and constant. Irrelevant events are removed from memory as soon as they are "consumed" (i.e., propagated upward or triggered).

The second experiment consists of compiling the real end-of-day market stock values as event triggers for the IBM company (in the last 40 years, approx. 10K stock ticks). Each tuple has: the date, the opening and closing price, and the low and the high for the day. A composed event is triggered when the opening price is higher then the closing price, see rule (5). The execution time was 9.408sec, which is rather worse performance (approx. 1000 events/second) in comparison to the first experiment. The reason is twofold. First, in the latter experiment we deal with real stock events (i.e., with a set of data carried with each events). This data is indexed on the first attribute in XSB, which is inefficient for our experiment. Second, as real stock events are represented as predicates (not as propositions, see rules (4)), their processing is more complex.

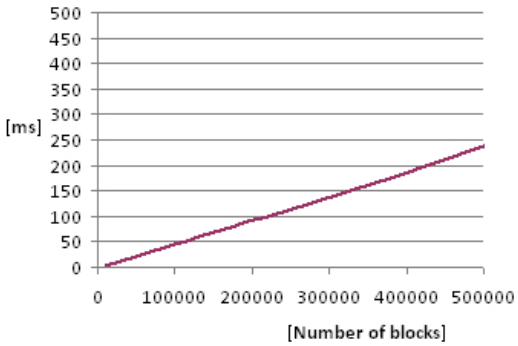

Figure 3: The event throughput for our experiments

$$ce3(Date) : -stock('IBM', Volume, Date, Open, Close, Low, High) * Open > Close. \quad (5)$$

We have presented first measurement results of our implementation. Even though there is a lot of room for improvements and optimizations, preliminary results show that logic-based event processing has capability to handle significant amount of events in reasonable time. Taking into account its strength (i.e., inference capability), it promises a powerful approach for combining *deductive* and *temporal* capabilities in a unified framework, yet with good run-time characteristics.

Further on, we have been working on a use case where data from Google Finance[4] service are converted to real-time events. These events are fed into ETALIS which then computes complex events of interest. Apart from event detection, deductive capabilities of ETALIS allows for exploring relationship between those complex events. This use case will also be demonstrated at DEBS 2009.

3. ACKNOWLEDGMENT

We thank Jia Ding and Ahmed Khalil Hafsi for their help in the implementation and testing of the prototype.

4. REFERENCES

[1] D. Anicic, P. Fodor, R. Stühmer, and N. Stojanovic. Efficient logic-based complex event processing and reactivity handling. In *Technical Report*, 2009. `http://code.google.com/p/etalis/wiki/TechnicalReport?ts=1241893834&updated=TechnicalReport`.

[3] $T_1 = T_2$ holds for any atomic event $a([T_1, T_2])$.

[4] Google Finance: http://www.google.com/finance?q=goog